Domain-specific Model Editors with Model Completion

Sagar Sen, Benoit Baudry, Hans Vangheluwe

Domain-specific Model Editors with Model Completion

Sagar Sen[1] Benoit Baudry[2]

IRISA/INRIA,
Campus universitaire de Beaulieu
35042 Rennes,France

Hans Vangheluwe[3]

School of Computer Science,
McGill University
Montreal,Quebec, Canada H3A2A7

Abstract

Today, integrated development environments such as Eclipse allow users to write programs quickly by presenting a set of recommendations for code completion. Similarly, word processing tools such as Microsoft Word present corrections for grammatical errors in sentences. Both of these existing systems use a set of constraints expressed in the form of a grammar to restrict/correct the user. Taking this idea further, in this paper we present an integrated software system capable of generating recommendations for model completion of partial models built in arbitrary domain specific model editors. We synthesize the model editor equipped with automatic completion from a modelling language's declarative specification consisting of a meta-model and constraints on it along with a visual syntax. The automatic completion feature is powered by a Prolog engine whose input is a constraint logic program derived from some models. The input logic program is obtained by a model transformation from models in multiple languages: the meta-model (as a class diagram), constraints on it (as constraint logic clauses), and a partial model (in the domain specific language). The Prolog engine solves the generated logic program and the solution(if there is one) is returned to the model editor as a set of recommendations for properties of the partial model. We incorporate automatic completion in the generative tool AToM^3 and use SWI-Prolog for constraint representation and satisfaction. We present examples using an illustrative visual language of Finite State Machines.

Keywords: constraint logic programming, meta-model, model editor, declarative specification, partial model, AToM^3

1 Introduction

Generative modelling tools such as AToM^3 (A Tool for Multiformalism Meta-modelling) [3],GME(Generic Modelling Environment)[5], GMF (Eclipse Graphical Modelling Framework)[4] can synthesize a domain specific visual model editor from

[1] Email: ssen@irisa.fr
[2] Email: bbaudry@irisa.fr
[3] Email: hv@cs.mcgill.ca

a declarative specification of a domain specific modelling language. A declarative specification consists of a meta-model, a set of constraints on all possible instances (or models) of the meta-model, and a visual syntax that describes how language elements(objects and relationships) manifesting the model editor. The designer of a model uses this model editor to construct a model on a canvas. This is analogous to a using an integrated development environment(IDE) to enter a program or a word processor to enter sentences. However, IDEs such as Eclipse present recommendations for completing a program statement when possible based on its grammar and existing libraries [2]. Similarly, Microsoft Word presents grammatical correction recommendations if a sentence does not conform to natural language grammar. Can we extrapolate similar technology for partial models constructed in a model editor for a domain specific modelling language(DSML)?

The major difficulty for providing completion capabilities in model editors is to integrate heterogeneous sources of knowledge in the computation of the possible solutions for completion. The completion algorithm must take into account the concepts defined in the meta-model for the DSML, the constraints expressed on this meta-model and the partial model built by a domain expert. The difficulty is that these three sources of knowledge are obviously related(they refer to the same concepts) but are expressed in different languages, sometimes in different files, and in most cases by different people and at different moments in the development cycle as they are separable concerns. These constraints are expressed in different languages and can come from multiple paradigms. For instance, in a electrical system model the resistance of a resistor can be constrained both by a low-level circuit model based on Kirchoff's laws and a high-level business process constraint that wants to have low-energy consumption at the resistor. The goal is to integrate these constraints under one common language.

In this paper, we propose an automatic transformation from all these sources of knowledge to a *constraint logic program* (CLP).The generated program can then be fed in a Prolog engine that provides the possible solutions for completing the model. Our transformation is integrated in the software tool AToM^3. The meta-model for a DSML is built directly in AToM^3's model editor using its class diagram formalism. The constraints on this meta-model are defined with Prolog in a separate file. Using this information and a description of the concrete visual syntax(specified in an icon editor) for a modelling language, AToM^3 synthesizes a visual model editor for the DSML. The partial model can be built and edited in the generated model editor and the designer can ask for recommendations for possible completions. The closest implementation in the literature is that of the GEMS tool [7]. In their work the detailed process of transforming a class-diagram based meta-model to a Prolog knowledge base is not discussed. Also, we clearly define the model completion problem boundaries and restrict ourselves to finite-domain constraint logic programs.

An overview of our methodology is presented in Section 2. In Section 3 we present how domain specific modelling languages are specified and model editors for them are synthesized in MDE using meta-models, constraints and visual syntax. We also present in Section 3 an example of a partial model and a complete model in our chosen domain. Using the meta-model,constraints, and partial model we present the transformation to a constraint logic program in Section 4. We present

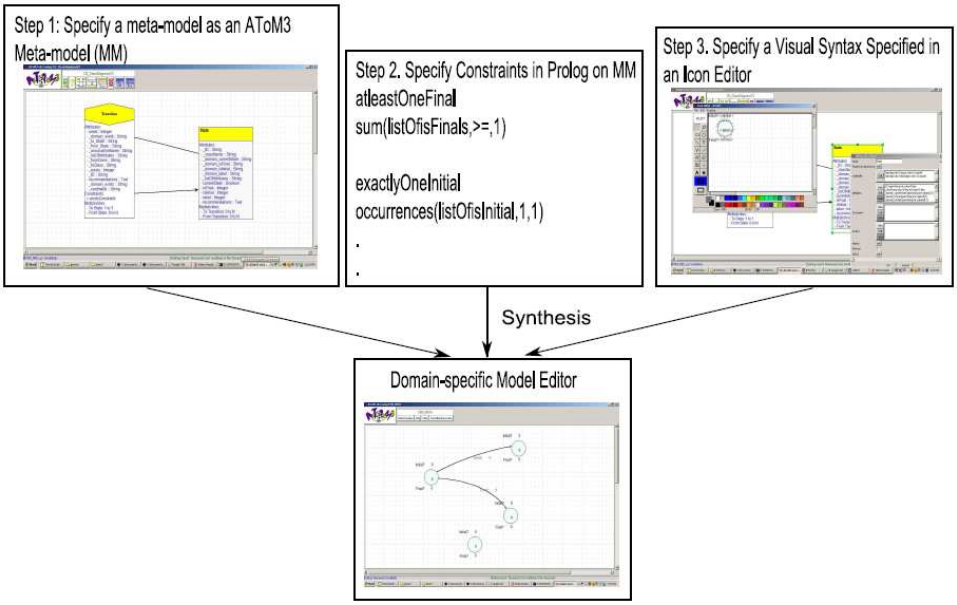

Fig. 1. Steps Taken by a DSML Designer to Synthesize a Model Editor in AToM^3

examples of model completion recommendations generated for partial models in Section 5. We conclude in Section 6 with limitations of our work and we layout future directions.

2 Methodology Overview

Synthesis of model editors elicit the involvement of several different experts and users. We identify the involvement of language designers, domain experts or users of a DSML, visual syntax designers for automatic synthesis of model editors.

- *Language Designers* interact with the domain experts to specify the concepts in a DSML in the form of a meta-model which is an AToM^3 class diagram [4] (meta-model from now on). Next, the designer specifies a set of Prolog clauses on the properties defined in the meta-model. We use SWI-Prolog for constraint representation.
- *Visual Syntax Designers* construct annotated icons that represent the different concepts in the meta-model. The icon for a class may be annotated with its property values. In Figure 1 we summarize how the meta-model, constraints,and visual syntax is used to synthesize a model editor for a DSML.
- *Domain Experts and Users* build models in the model editor that is synthesized from the meta-model, constraints, and visual syntax specifications. They also help the language designer define the concepts in the meta-model.

A domain expert uses the synthesized model editor to build models. He creates a model by inserting objects and building relationships between objects. He/she also sets values for properties. The model is simply a graph or a partial model until

[4] AToM^3 class diagram is a subset of UML class diagram for meta-modelling and has sufficient expressiveness for *bootstrapping*

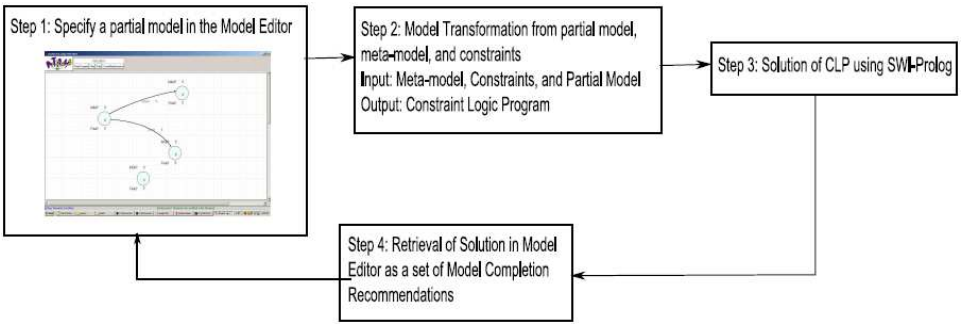

Fig. 2. Model Completion Outline

it conforms to its modelling language by satisfying all the constraints imposed on the modelling language. Manually performing such a task can be extremely tedious and sometimes impossible due to the size of the domains of model properties and complexity of constraints.

To automate the completion of a partial model we introduce a model transformation to construct a generative algorithm from the knowledge provided in the meta-model and constraints. The algorithm takes a partial model as input and generates a constraint logic program. This transformation is integrated into AToM^3 so that it can be used for completion of models in any domain specific language.

Logic programming tool developers have built Prolog compilers [6] that can perform computer algebra and constraint satisfaction on an input constraint logic program. Such a Prolog compiler is invoked by AToM^3 and the synthesized CLP is solved and the results (if they exist) are returned to the model editor as recommendations. We use SWI-Prolog [6] for compiling the constraint logic program. In Figure 2 we outline how we complete partial models in AToM^3 .

Now that we have outlined our overall methodology we go ahead and study each aspect of the methodology in detail leading to examples that illustrate the working of the idea. We illustrate our methodology using the guiding example of a Finite State Machine (FSM) modelling language.

3 Specifying a Domain Specific Modelling Language

In this section we explain the steps taken to declaratively specifying a domain specific modelling language. We use Finite State Machines (FSM) as a running example for a modelling language. A FSM modelling language is a visual language with circles representing states and directed arrows representing transitions between states. To define a modelling language and to generate visual model editor from it requires three inputs:

(i) A Meta-model as an AToM^3 class diagram

(ii) A Set of Prolog Constraints on the meta-model

(iii) A Visual Syntax

We briefly describe these in the following sub-sections.

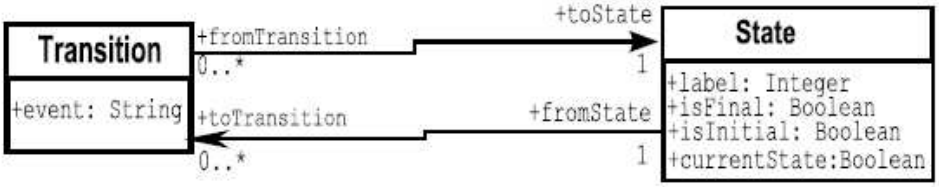

Fig. 3. The Finite State Machine Meta-model

Table 1
Domains for Primitive Datatypes

Type	**Domain**
Boolean	$\{0, 1\}$
Integer	$\{MinInt, .., MaxInt\}$
String	$\{$"a", "b", "c", "$event1$", .., $\}$

3.1 Meta-model

A model consists of objects and relationships between them. The meta-model of the modelling language specifies the types of all the objects and their possible inter-relationships. The type of an object is referred to as a class. The meta-model for the FSM modelling language is presented in Figure 3. The classes in the meta-model are **State** and **Transition**.

In this paper we use the class diagram formalism in AToM^3 for specifying a meta-model. The class diagram formalism can specify itself and hence exhibits the property of *bootstrapping*. We use the visual language notation of class diagrams to specify the meta-model for the FSM modelling language in Figure 3.

Each class in the meta-model has *properties*. A property is either an *attribute* or a *reference*. An attribute is of primitive type which is either Integer, String, or Boolean. For instance, the attributes of the class **State** are isInitial and isFinal both of which are of primitive type Boolean. An example domain of values for the primitive attributes is given in Table 1. The String variable can be a finite set consisting of a null string, and finite length strings that specify a set of strings. In this paper, we consider a finite domain for each attribute. The domain is specified in the meta-model and all the models that are instances of the meta-model know of the domain for each attribute.

Describing the state of a class of objects with only primitive attributes is not sufficient in many cases. Modelling many real-world systems elicits the need to model complex relationships such as modelling that an object contains another set of objects or an object is related to another finite set of objects. This set of related objects is constrained by a *cardinality*. When a class is related to another class, the related classes refer to each other via *references*. For instance, in Figure 3 the classes **State** and **Transition** refer to each other via references annotated with uni-directional relationships. The cardinality constraints are also annotated with the relationship.

Apart from attributes and references, objects can inherit properties from other classes. The attributes and references of a class called a super class are inherited by

derived classes. Similarly a derived class inherits the references in the super class. There is no inheritance in our FSM meta-model, nevertheless we consider transformation of inheritance relationships in the transformation presented in Section 4.

3.2 Constraints on Meta-model

Constraints on a meta-model are not always conveniently specified using diagrams. They are better expressed in a textual constraint language who's semantics has no side-effect (does not change the state of an object or structure of the model) on the meta-model or its instances (models). The OMG standard for constraint specification is Object Constraint Language (OCL) however in our current work we use constraint logic programming clauses in the form of Prolog statements. These constraints are initially specified on meta-model properties. The transformation generates a set of constraints on a lists of properties (those influencing the constraint) in the partial model.

We use the CLP bounds library to specify constraints on properties with finite domain. There are several predicates in the standard Prolog library. For instance, one of the constraints for the FSM modelling language is:

- atLeastOneFinalState
  Variables: listOfisFinal is the list of all isFinal attributes for all states in the model.
  Prolog Constraint: sum(listOfisFinal,>=,1)
  Explanation: The attribute isFinal is a boolean and the list of isFinals contains the values of all attributes in the partial model. The constraint ensures that the sum of the isFinals is greater than or equal to 1. This enforces the constraint that there is at least one final state.

Other constraints include exactly one initial state, and a unique label for a state object. To define constraints for arbitrary DSMLs we point the reader to the SWI-Prolog reference manual [6]. The language and the libraries have been developed for two decades and we have a large repository of constraints to work with including facility to use a foreign language to define an arbitrary boolean function. Prolog has powerful mechanisms such as *domain reduction*. For instance, constraint alldifferent(listOfVariables) ensures the automatic reduction in the domain of variables in listOfVariables such that the each variable in the list has a domain with values not in the domain of the others. The textual specification of constraints is typically specified in an different file from the class diagram meta-model itself.

3.3 Visual Syntax

The final step(in specifying a DSML for synthesizing a model editor) we take is to specify the concrete visual syntax of the class of objects in the meta-model. The visual syntax specifies what an object looks like on a 2D canvas. An icon editor in AToM^3 is used to specify the visual syntax of the classes in the meta-model.

An icon editor is used to specify the visual syntax of meta-model concepts such as classes and relationships. The icon for **State** is a circle annotated with three of

its attributes(isFinal, isInitial, and label). The connectors in the diagram are points of connection between **State** objects and **Transition** objects.

The visual syntax can also be dynamically changed based on the properties of the model for example. In an iconic visual modelling language such FSM the first step taken in specifying a visual syntax is drawing an icon that represents a class of objects. If needed it is annotated with text and its properties. Connectors are added to the visual object so that it can be connected to other objects if they are related.

4 Transformation from Declarative Specification and a Partial Model to CLP

We present the transformation of the different parts of a partial model to a CLP using the meta-model and constraints as input. The essential idea to generate CLPs constitutes the following steps:

(i) Create variables to represent properties of a partial model

(ii) Define a domain on these variables.

(iii) Define constraints on these variables.

(iv) Finally, insert the label(SetOfVariables) clause to perform back-tracking search.

We associate a *finite domain* for each variable in the constraint logic program (CLP) hence, making it a *constraint logic program in finite domain* (CLP(FD)).We use the clp_bounds library in SWI-Prolog to express domains and constraints in CLP(FD). We generate a conjunction of constraints in Prolog. The conjunction is given by a **,** operator. Finally, we insert the label predicate at the end of the program to perform back-tracking to find the value assignment/labeling of variables so as to generate completions for the partial model.

4.1 Transforming an Object

We now discuss how objects in a partial model are transformed to CLP. We illustrate this with a concrete example to enhance the reader's understanding. Consider the object shown in Figure 4. It is a **State** object. The attributes of a **State** object are isFinal, isInitial, currentState, and label. Each attribute also has a domain. The attributes isFinal, isInitial and currentState in the **State** object has a boolean domain of $[0, 1]$. The label attribute has an integer domain of $[0, 1, 2, .., MaxNumberOfStates]$.

In the transformation first each attribute has a unique identity which is given by OwningObjectName_attributeName. This unique ID is used to create a variable and is added to a list of variables in the CLP. If Model is the set of variables in the partial model. Then the variable State0_isFinal is included in this list:

Model=[..., State0_isFinal,...],

Next, we associate a domain with a variable already included in the list of model variables. This is done using the member Prolog predicate. For instance, the domain

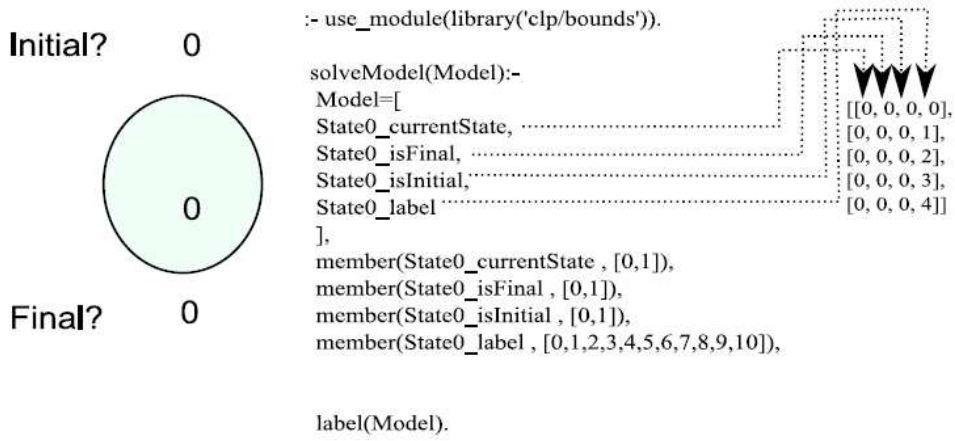

Fig. 4. (a) Object (example of a **State** object (b) Generated CLP code (c) Five Prolog solutions for each variable

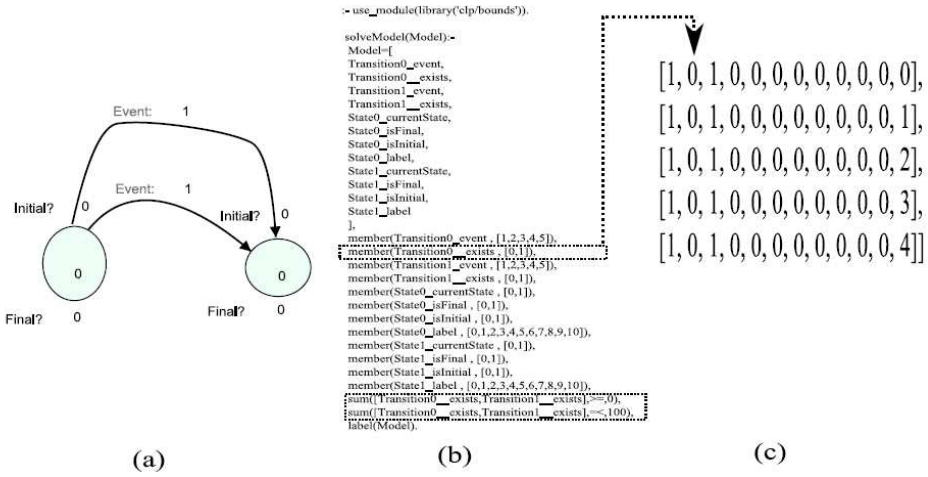

Fig. 5. (a) Association between **State** objects and a **Transition** (b) Generated CLP code (c) Cardinality constraint determines the value of an existence variable.

for the variable State0_isFinal is manifested in Prolog as follows:

member(State0_isFinal,[0,1])

We obtain the domain information for an attribute from the meta-model of the modelling language.

The CLP code generated for a **State** object is shown in Figure 4 (b). Solving the Prolog program gives a set of arrays with the result for the value assignment of each variable. This is shown in Figure 4 (c).

4.2 Transforming an Association

Next, we consider the transformation of an association in partial model to Prolog clauses. Consider the associations in Figure 5 (a). Two **State** objects are connected by two **Transition** objects. The existence of these relationships is determined by boolean *existence variables* such as Transition0_exists and Transition1_exists. In general, these variables are synthesized for all association in the partial model. We

obtain the cardinality constraint for each association from the meta-model. In the partial model we look for all associations with the same source and destination object. We impose a cardinality constraint on all associations with the same source and destination. We synthesize two Prolog clauses to impose the cardinality constraint on a list of existence variables for the associations with the same source and destination. The example in Figure 5 (a) has its code generated in Figure 4 (b).

The cardinality constraints are imposed as sum of existence variables as follows:

sum([Transition0 exists,Transition1 exists],>=,0),
sum([Transition0 exists,Transition1 exists],=<,100),

The solution obtained for completing the partial model is shown in Figure 5 (c).

4.3 Generating Constraints

Finally, we insert constraints defined on the meta-model. A constraint C is expressed on properties p1,p2,...pN of a meta-model MM. In a partial model we identify all properties that are constrained by C and generate a list of variables (those already generated as described in Sections 4.1 and 4.2). The constraint on the meta-model itself cannot be executed. The constraint of the partial model is handled by the Prolog compiler.

For instance, to ensure that every **State** object in the partial model has a unique label we generate the following constraint which is added as a conjunction to the constraints already generated:

all different([State0_label,State1_label,..])

The all_different clause ensures that he value of each element in the list it receives as input is unique.

5 A Running Example

In this section, we present an example of a partial model that we use to generate model completion recommendations. In Figure 6 we present a partial model with two generated recommendations. For the same partial model we performed more tests. We generate 5 model completion recommendations. We randomly shuffle the domain constraints in the generated CLP. The shuffling changes the priority order in which values for properties are chosen by Prolog and has an effect on the result of model completion. We do not study this variability in detail. However, we present the time taken for generating 5 recommendations in Table 2.

For our example the time taken to generate a solution for the modelis reasonably acceptable with an average of 2.5 seconds. A large portion of the time taken involves pre-processing of the problem by the Prolog compiler. The rest of the time is taken to find value assignments for constraint satisfaction.

6 Conclusion

In this paper we present a framework for generating model completion recommendations in model editors. We illustrate our approach with the simple example of the

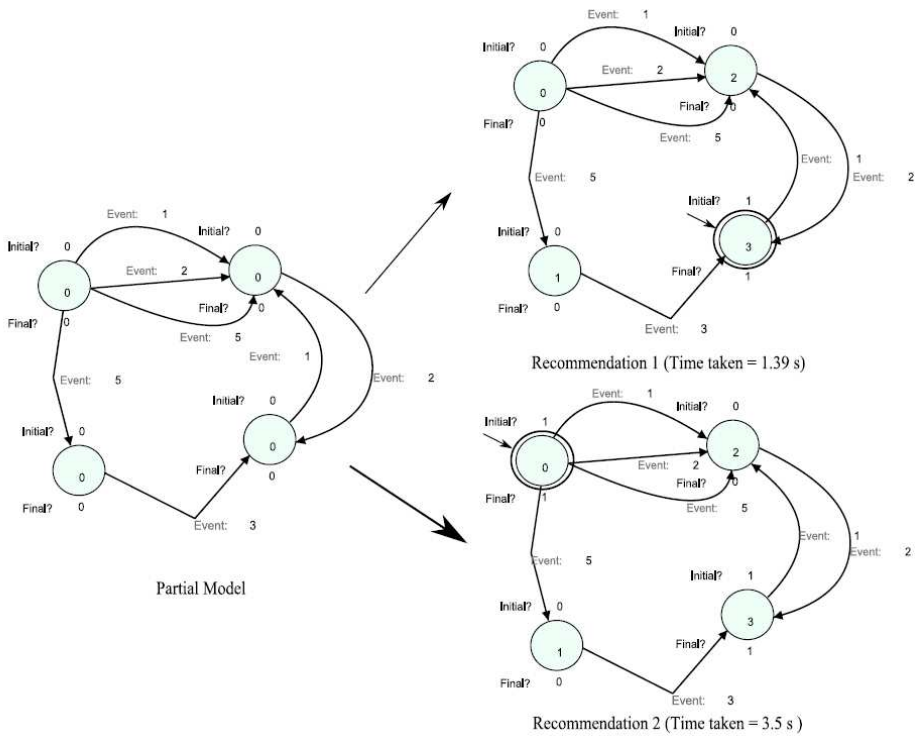

Fig. 6. A Partial Model and Two Proposed Recommendations

Table 2
Generated Recommendations

Recommendation	CPU Time
1	1.3
2	0.55
3	3.34
4	3.50
5	3.72

FSM modelling language. At present we specify the meta-model as an AToM^3 class diagram (which is subset of UML class diagrams with sufficient facility for *bootstrapping*). Constraints on the meta-model are directly specified in Prolog. We also demonstrate, using a reasonably complex example, the working of our approach. However, there is room for several improvements.

Currently we only support constraint satisfaction of constraints from the meta-model and constraints of the modelling language. We wish to extend this by introducing *user-specific objective functions* and other constraints such as model transformation pre-conditions. This could lead to synthesis of interesting models for tasks such as model transformation testing and design space exploration. It would also be interesting to see how constraints from multiple paradigms can co-exist in the same environment and how they can be solved to get meaningful results.

Also, we start from a partial model with a fixed number of objects. In other

words the *dimensionality* given by the object space is fixed. We plan to separate the notion of object space and property space to allow the synthesis of objects that add to a partial model before we go ahead and find values for proper ties as explained in this article. He have not rigoursly tested our approach for scalability. With the increase in the number of objects the Prolog engine takes longer to find a solution. A way to speed up would be to reduce the number of elements in the domain of properties in the partial model. Another approach would be divide the model into components and complete each component before integration.

Finally, we plan to use high-level constraints to formally communicate knowledge between modelling domains(multi-paradigm modelling) or scientific knowledge in general.

References

[1] Krysztof R. Apt, and Mark G. Wallace, *Constraint Logic Programming with ECLiPSe*, Cambridge University Press(2007).

[2] Andrew J. Ko, Htet Htet Aung and Brad A. Myers, *Design requirements for more flexible structured editors from a study of programmers text editing,* CHI 05, URL: http://portal.acm.org/citation.cfm?id=1056808.1056965.

[3] Hans Vangheluwe and Juan de Lara, *Domain-Specific Modelling with* AToM^3.In Juha-Pekka Tolvanen, Jonathan Sprinkle, and Matti Rossi, editors,The 4th OOPSLA Workshop on Domain-Specific Modeling, page 8 pp., October 2004. Vancouver, Canada.

[4] Karsten Ehrig, Claudia Ermel, and Stegan Hansgen, *Generation of visual editors as eclipse plug-ins* in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 134 -143 , 2005

[5] R Ledeczi, A., Bakay, A.; Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. and Karsai, G. *Composing Domain-Specific Design Environments* in Computer, 44-51, 2001

[6] Jan Wielemaker, *SWI-Prolog 5.6.35 Reference Manual*,2007 URL: http://gollem.science.uva.nl/SWI-Prolog/Manual/

[7] Jules White, Andrey Nechypurenko, Egon Wuchner, and Douglas C. Schmidt, *Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains*, Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems, October 23, 2006, Portland, Oregon.